自动引用计数（Automatic Reference Counting）

1.0 翻译：TimothyYe (https://github.com/TimothyYe) 校对：Hawstein (https://github.com/Hawstein)

2.0 翻译+校对：Channe (https://github.com/Channe)

2.1 翻译：Channe (https://github.com/Channe) 校对：shanks (http://codebuild.me)，Realank (https://github.com/Realank)，2016-01-23

2.2 翻译+校对：SketchK (https://github.com/SketchK) 2016-05-14

本页包含内容：

Swift 使用自动引用计数（ARC）机制来跟踪和管理你的应用程序的内存。通常情况下，Swift 内存管理机制会一直起作用，你无须自己来考虑内存的管理。ARC 会在类的实例不再被使用时，自动释放其占用的内存。

然而在少数情况下，为了能帮助你管理内存，ARC 需要更多的，代码之间关系的信息。本章描述了这些情况，并且为你示范怎样才能使 ARC 来管理你的应用程序的所有内存。

> 注意
> 引用计数仅仅应用于类的实例。结构体和枚举类型是值类型，不是引用类型，也不是通过引用的方式存储和传递。

自动引用计数的工作机制

当你每次创建一个类的新的实例的时候，ARC 会分配一块内存来储存该实例信息。内存中会包含实例的类型信息，以及这个实例所有相关的存储型属性的值。

此外，当实例不再被使用时，ARC 释放实例所占用的内存，并让释放的内存能挪作他用。这确保了不再被使用的实例，不会一直占用内存空间。

然而，当 ARC 收回和释放了正在被使用中的实例，该实例的属性和方法将不能再被访问和调用。实际上，如果你试图访问这个实例，你的应用程序很可能会崩溃。

为了确保使用中的实例不会被销毁，ARC 会跟踪和计算每一个实例正在被多少属性，常量和变量所引用。哪怕实例的引用数为1，ARC都不会销毁这个实例。

为了使上述成为可能，无论你将实例赋值给属性、常量或变量，它们都会创建此实例的强引用。之所以称之为“强”引用，是因为它会将实例牢牢地保持住，只要强引用还在，实例是不允许被销毁的。

自动引用计数实践

下面的例子展示了自动引用计数的工作机制。例子以一个简单的 `Person` 类开始，并定义了一个叫 `name` 的常量属性：

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

`Person` 类有一个构造函数，此构造函数为实例的 `name` 属性赋值，并打印一条消息以表明初始化过程生效。 `Person` 类也拥有一个析构函数，这个析构函数会在实例被销毁时打印一条消息。

接下来的代码片段定义了三个类型为 `Person?` 的变量，用来按照代码片段中的顺序，为新的 `Person` 实例建立多个引用。由于这些变量是被定义为可选类型（ `Person?` ，而不是 `Person` ），它们的值会被自动初始化为 `nil` ，目前还不会引用到 `Person` 类的实例。

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

现在你可以创建 Person 类的新实例，并且将它赋值给三个变量中的一个：

```
reference1 = Person(name: "John Appleseed")
// prints "John Appleseed is being initialized”
```

应当注意到当你调用 Person 类的构造函数的时候，“John Appleseed is being initialized”会被打印出来。由此可以确定构造函数被执行。

由于 Person 类的新实例被赋值给了 reference1 变量，所以 reference1 到 Person 类的新实例之间建立了一个强引用。正是因为这一个强引用，ARC 会保证 Person 实例被保持在内存中不被销毁。

如果你将同一个 Person 实例也赋值给其他两个变量，该实例又会多出两个强引用：

```
reference2 = reference1
reference3 = reference1
```

现在这一个 Person 实例已经有三个强引用了。

如果你通过给其中两个变量赋值 nil 的方式断开两个强引用（包括最先的那个强引用），只留下一个强引用， Person 实例不会被销毁：

```
reference1 = nil
reference2 = nil
```

在你清楚地表明不再使用这个 Person 实例时，即第三个也就是最后一个强引用被断开时，ARC 会销毁它：

```
reference3 = nil
// 打印 “John Appleseed is being deinitialized”
```

类实例之间的循环强引用

在上面的例子中，ARC 会跟踪你所新创建的 Person 实例的引用数量，并且会在 Person 实例不再被需要时销毁它。

然而，我们可能会写出一个类实例的强引用数永远不能变成 0 的代码。如果两个类实例互相持有对方的强引用，因而每个实例都让对方一直存在，就是这种情况。这就是所谓的循环强引用。

你可以通过定义类之间的关系为弱引用或无主引用，以替代强引用，从而解决循环强引用的问题。具体的过程在解决类实例之间的循环强引用中有描述。不管怎样，在你学习怎样解决循环强引用之前，很有必要了解一下它是怎样产生的。

下面展示了一个不经意产生循环强引用的例子。例子定义了两个类： Person 和 Apartment ，用来建模公寓和它其中的居民：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
```

```
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

每一个 Person 实例有一个类型为 String ， 名字为 name 的属性，并有一个可选的初始化为 nil 的 apartment 属性。 apartment 属性是可选的，因为一个人并不总是拥有公寓。

类似的，每个 Apartment 实例有一个叫 unit ， 类型为 String 的属性，并有一个可选的初始化为 nil 的 tenant 属性。 tenant 属性是可选的，因为一栋公寓并不总是有居民。

这两个类都定义了析构函数，用以在类实例被析构的时候输出信息。这让你能够知晓 Person 和 Apartment 的实例是否像预期的那样被销毁。

接下来的代码片段定义了两个可选类型的变量 john 和 unit4A ，并分别被设定为下面的 Apartment 和 Person 的实例。这两个变量都被初始化为 nil ，这正是可选的优点：

```
var john: Person?
var unit4A: Apartment?
```

现在你可以创建特定的 `Person` 和 `Apartment` 实例并将赋值给 `john` 和 `unit4A` 变量：

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

在两个实例被创建和赋值后，下图表现了强引用的关系。变量 `john` 现在有一个指向 `Person` 实例的强引用，而变量 `unit4A` 有一个指向 `Apartment` 实例的强引用：

现在你能够将这两个实例关联在一起，这样人就能有公寓住了，而公寓也有了房客。注意感叹号是用来展开和访问可选变量 `john` 和 `unit4A` 中的实例，这样实例的属性才能被赋值：

```
john!.apartment = unit4A
unit4A!.tenant = john
```

在将两个实例联系在一起之后，强引用的关系如图所示：

不幸的是，这两个实例关联后会产生一个循环强引用。 `Person` 实例现在有了一个指向 `Apartment` 实例的强引用，而 `Apartment` 实例也有了一个指向 `Person` 实例的强引用。因此，当你断开 `john` 和 `unit4A` 变量所持有的强引用时，引用计数并不会降为 `0` ，实例也不会被 ARC 销毁：

```
john = nil
unit4A = nil
```

注意，当你把这两个变量设为 `nil` 时，没有任何一个析构函数被调用。循环强引用会一直阻止 `Person` 和 `Apartment` 类实例的销毁，这就在你的应用程序中造成了内存泄漏。

在你将 `john` 和 `unit4A` 赋值为 `nil` 后，强引用关系如下图：

`Person` 和 `Apartment` 实例之间的强引用关系保留了下来并且不会被断开。

解决实例之间的循环强引用

Swift 提供了两种办法用来解决你在使用类的属性时所遇到的循环强引用问题：弱引用（weak reference）和无主引用（unowned reference）。

弱引用和无主引用允许循环引用中的一个实例引用另外一个实例而不保持强引用。这样实例能够互相引用而不产生循环强引用。

对于生命周期中会变为 `nil` 的实例使用弱引用。相反地，对于初始化赋值后再也不会被赋值为 `nil` 的实例，使用无主引用。

弱引用

弱引用不会对其引用的实例保持强引用，因而不会阻止 ARC 销毁被引用的实例。这个特性阻止了引用变为循环强引用。声明属性或者变量时，在前面加上 `weak` 关键字表明这是一个弱引用。

在实例的生命周期中，如果某些时候引用没有值，那么弱引用可以避免循环强引用。如果引用总是有值，则可以使用无主引用，在无主引用中有描述。在上面 `Apartment` 的例子中，一个公寓的生命周期中，有时是没有"居民"的，因此适合使用弱引用来解决循环强引用。

> 注意
> 弱引用必须被声明为变量，表明其值能在运行时被修改。弱引用不能被声明为常量。

因为弱引用可以没有值，你必须将每一个弱引用声明为可选类型。在 Swift 中，推荐使用可选类型描述可能没有值的类型。

因为弱引用不会保持所引用的实例，即使引用存在，实例也有可能被销毁。因此，ARC 会在引用的实例被销毁后自动将其赋值为 `nil` 。你可以像其他可选值一样，检查弱引用的值是否存在，你将永远不会访问已销毁的实例的引用。

下面的例子跟上面 `Person` 和 `Apartment` 的例子一致，但是有一个重要的区别。这一次， `Apartment` 的 `tenant` 属性被声明为弱引用：

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
```

```
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

然后跟之前一样，建立两个变量（ `john` 和 `unit4A` ）之间的强引用，并关联两个实例：

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

现在，两个关联在一起的实例的引用关系如下图所示：

`Person` 实例依然保持对 `Apartment` 实例的强引用，但是 `Apartment` 实例只持有对 `Person` 实例的弱引用。这意味着当你断开 `john` 变量所保持的强引用时，再也没有指向 `Person` 实例的强引用了：

由于再也没有指向 `Person` 实例的强引用，该实例会被销毁：

```
john = nil
// 打印 “John Appleseed is being deinitialized”
```

唯一剩下的指向 `Apartment` 实例的强引用来自于变量 `unit4A` 。如果你断开这个强引用，再也没有指向 `Apartment` 实例的强引用了：

由于再也没有指向 `Apartment` 实例的强引用，该实例也会被销毁：

```
unit4A = nil
// 打印 “Apartment 4A is being deinitialized”
```

上面的两段代码展示了变量 `john` 和 `unit4A` 在被赋值为 `nil` 后， `Person` 实例和 `Apartment` 实例的析构函数都打印出“销毁”的信息。这证明了引用循环被打破了。

> 注意
> 在使用垃圾收集的系统里，弱指针有时用来实现简单的缓冲机制，因为没有强引用的对象只会在内存压力触发垃圾收集时才被销毁。但是在 ARC 中，一旦值的最后一个强引用被移除，就会被立即销毁，这导致弱引用并不适合上面的用途。

无主引用

和弱引用类似，无主引用不会牢牢保持住引用的实例。和弱引用不同的是，无主引用是永远有值的。因此，无主引用总是被定义为非可选类型（non-optional type）。你可以在声明属性或者变量时，在前面加上关键字 `unowned` 表示这是一个无主引用。

由于无主引用是非可选类型，你不需要在使用它的时候将它展开。无主引用总是可以被直接访问。不过 ARC 无法在实例被销毁后将无主引用设为 `nil` ，因为非可选类型的变量不允许被赋值为 `nil` 。

> 注意
>
> 如果你试图在实例被销毁后，访问该实例的无主引用，会触发运行时错误。使用无主引用，你必须确保引用始终指向一个未销毁的实例。
>
> 还需要注意的是如果你试图访问实例已经被销毁的无主引用，Swift 确保程序会直接崩溃，而不会发生无法预期的行为。所以你应当避免这样的事情发生。

下面的例子定义了两个类， `Customer` 和 `CreditCard` ，模拟了银行客户和客户的信用卡。这两个类中，每一个都将另外一个类的实例作为自身的属性。这种关系可能会造成循环强引用。

`Customer` 和 `CreditCard` 之间的关系与前面弱引用例子中 `Apartment` 和 `Person` 的关系略微不同。在这个数据模型中，一个客户可能有或者没有信用卡，但是一张信用卡总是关联着一个客户。为了表示这种关系， `Customer` 类有一个可选类型的 `card` 属性，但是 `CreditCard` 类有一个非可选类型的 `customer` 属性。

此外，只能通过将一个 `number` 值和 `customer` 实例传递给 `CreditCard` 构造函数的方式来创建 `CreditCard` 实例。这样可以确保当创建 `CreditCard` 实例时总是有一个 `customer` 实例与之关联。

由于信用卡总是关联着一个客户，因此将 `customer` 属性定义为无主引用，用以避免循环强引用：

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}
```

```
class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}
```

> 注意
>
> `CreditCard` 类的 `number` 属性被定义为 `UInt64` 类型而不是 `Int` 类型，以确保 `number` 属性的存储量在 32 位和 64 位系统上都能足够容纳 16 位的卡号。

下面的代码片段定义了一个叫 `john` 的可选类型 `Customer` 变量，用来保存某个特定客户的引用。由于是可选类型，所以变量被初始化为 `nil` ：

```
var john: Customer?
```

现在你可以创建 `Customer` 类的实例，用它初始化 `CreditCard` 实例，并将新创建的 `CreditCard` 实例赋值为客户的 `card` 属性：

```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

在你关联两个实例后，它们的引用关系如下图所示：

`Customer` 实例持有对 `CreditCard` 实例的强引用，而 `CreditCard` 实例持有对 `Customer` 实例的无主引用。

由于 `customer` 的无主引用，当你断开 `john` 变量持有的强引用时，再也没有指向 `Customer` 实例的强引用了：

由于再也没有指向 `Customer` 实例的强引用，该实例被销毁了。其后，再也没有指向 `CreditCard` 实例的强引用，该实例也随之被销毁了：

```
john = nil
// 打印 “John Appleseed is being deinitialized”
// 打印 ”Card #1234567890123456 is being deinitialized”
```

最后的代码展示了在 `john` 变量被设为 `nil` 后 `Customer` 实例和 `CreditCard` 实例的构造函数都打印出了“销毁”的信息。

无主引用以及隐式解析可选属性

上面弱引用和无主引用的例子涵盖了两种常用的需要打破循环强引用的场景。

`Person` 和 `Apartment` 的例子展示了两个属性的值都允许为 `nil`，并会潜在的产生循环强引用。这种场景最适合用弱引用来解决。

`Customer` 和 `CreditCard` 的例子展示了一个属性的值允许为 `nil`，而另一个属性的值不允许为 `nil`，这也可能会产生循环强引用。这种场景最适合通过无主引用来解决。

然而，存在着第三种场景，在这种场景中，两个属性都必须有值，并且初始化完成后永远不会为 `nil`。在这种场景中，需要一个类使用无主属性，而另外一个类使用隐式解析可选属性。

这使两个属性在初始化完成后能被直接访问（不需要可选展开），同时避免了循环引用。这一节将为你展示如何建立这种关系。

下面的例子定义了两个类，`Country` 和 `City`，每个类将另外一个类的实例保存为属性。在这个模型中，每个国家必须有首都，每个城市必须属于一个国家。为了实现这种关系，`Country` 类拥有一个 `capitalCity` 属性，而 `City` 类有一个 `country` 属性：

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}
```

```
class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

为了建立两个类的依赖关系，`City` 的构造函数接受一个 `Country` 实例作为参数，并且将实例保存到 `country` 属性。

`Country` 的构造函数调用了 `City` 的构造函数。然而，只有 `Country` 的实例完全初始化后，`Country` 的构造函数才能把 `self` 传给 `City` 的构造函数。（在两段式构造过程 (./14_Initialization.html#two_phase_initialization)中有具体描述）

为了满足这种需求，通过在类型结尾处加上感叹号（`City!`）的方式，将 `Country` 的 `capitalCity` 属性声明为隐式解析可选类型的属性。这意味着像其他可选类型一样，`capitalCity` 属性的默认值为 `nil`，但是不需要展开它的值就能访问它。（在隐式解析可选类型 (./01_The_Basics.html#implicityly_unwrapped_optionals)中有描述）

由于 `capitalCity` 默认值为 `nil`，一旦 `Country` 的实例在构造函数中给 `name` 属性赋值后，整个初始化过程就完成了。这意味着一旦 `name` 属性被赋值后，`Country` 的构造函数就能引用并传递隐式的 `self`。`Country` 的构造函数在赋值 `capitalCity` 时，就能将 `self` 作为参数传递给 `City` 的构造函数。

以上的意义在于你可以通过一条语句同时创建 `Country` 和 `City` 的实例，而不产生循环强引用，并且 `capitalCity` 的属性能被直接访问，而不需要通过感叹号来展开它的可选值：

```
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \(country.capitalCity.name)")
// 打印 “Canada's capital city is called Ottawa”
```

在上面的例子中，使用隐式解析可选值意味着满足了类的构造函数的两个构造阶段的要求。`capitalCity` 属性在初始化完成后，能像非可选值一样使用和存取，同时还避免了循环强引用。

闭包引起的循环强引用

前面我们看到了循环强引用是在两个类实例属性互相保持对方的强引用时产生的，还知道了如何用弱引用和无主引用来打破这些循环强引用。

循环强引用还会发生在当你将一个闭包赋值给类实例的某个属性，并且这个闭包体中又使用了这个类实例时。这个闭包体中可能访问了实例的某个属性，例如 `self.someProperty`，或者闭包中调用了实例的某个方法，例如 `self.someMethod()`。这两种情况都导致了闭包“捕获”`self`，从而产生了循环强引用。

循环强引用的产生，是因为闭包和类相似，都是引用类型。当你把一个闭包赋值给某个属性时，你是将这个闭包的引用赋值给了属性。实质上，这跟之前的问题是一样的——两个强引用让彼此一直有效。但是，和两个类实例不同，这次一个是类实例，另一个是闭包。

Swift 提供了一种优雅的方法来解决这个问题，称之为 闭包捕获列表（closure capture list）。同样的，在学习如何用闭包捕获列表打破循环强引用之前，先来了解一下这里的循环强引用是如何产生的，这对我们很有帮助。

下面的例子为你展示了当一个闭包引用了 `self` 后是如何产生一个循环强引用的。例子中定义了一个叫 `HTMLElement` 的类，用一种简单的模型表示 HTML 文档中的一个单独的元素：

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: Void -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

`HTMLElement` 类定义了一个 `name` 属性来表示这个元素的名称，例如代表段落的 “`p`”，或者代表换行的 “`br`”。`HTMLElement` 还定义了一个可选属性 `text`，用来设置 HTML 元素呈现的文本。

除了上面的两个属性，`HTMLElement` 还定义了一个 `lazy` 属性 `asHTML`。这个属性引用了一个将 `name` 和 `text` 组合成 HTML 字符串片段的闭包。该属性是 `Void -> String` 类型，或者可以理解为“一个没有参数，返回 `String` 的函数”。

默认情况下，闭包赋值给了 `asHTML` 属性，这个闭包返回一个代表 HTML 标签的字符串。如果 `text` 值存在，该标签就包含可选值 `text`；如果 `text` 不存在，该标签就不包含文本。对于段落元素，根据 `text` 是 “`some text`” 还是 `nil`，闭包会返回 `"<p>some text</p>"` 或者 `"<p />"`。

可以像实例方法那样去命名、使用 `asHTML` 属性。然而，由于 `asHTML` 是闭包而不是实例方法，如果你想改变特定 HTML 元素的处理方式的话，可以用自定义的闭包来取代默认值。

例如，可以将一个闭包赋值给 `asHTML` 属性，这个闭包能在 `text` 属性是 `nil` 时使用默认文本，这是为了避免返回一个空的 HTML 标签：

```
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\(heading.name)>"
}
print(heading.asHTML())
// 打印 “<h1>some default text</h1>”
```

> 注意
>
> `asHTML` 声明为 `lazy` 属性，因为只有当元素确实需要被处理为 HTML 输出的字符串时，才需要使用 `asHTML`。也就是说，在默认的闭包中可以使用 `self`，因为只有当初始化完成以及 `self` 确实存在后，才能访问 `lazy` 属性。

`HTMLElement` 类只提供了一个构造函数，通过 `name` 和 `text`（如果有的话）参数来初始化一个新元素。该类也定义了一个析构函数，当 `HTMLElement` 实例被销毁时，打印一条消息。

下面的代码展示了如何用 `HTMLElement` 类创建实例并打印消息：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// 打印 “<p>hello, world</p>”
```

> 注意
> 上面的 `paragraph` 变量定义为可选类型的 `HTMLElement` ，因此我们可以赋值 `nil` 给它来演示循环强引用。

不幸的是，上面写的 `HTMLElement` 类产生了类实例和作为 `asHTML` 默认值的闭包之间的循环强引用。循环强引用如下图所示：

实例的 `asHTML` 属性持有闭包的强引用。但是，闭包在其闭包体内使用了 `self` （引用了 `self.name` 和 `self.text` ），因此闭包捕获了 `self` ，这意味着闭包又反过来持有了 `HTMLElement` 实例的强引用。这样两个对象就产生了循环强引用。（更多关于闭包捕获值的信息，请参考值捕获 (./07_Closures.html#capturing_values)）。

> 注意
> 虽然闭包多次使用了 `self` ，它只捕获 `HTMLElement` 实例的一个强引用。

如果设置 `paragraph` 变量为 `nil` ，打破它持有的 `HTMLElement` 实例的强引用， `HTMLElement` 实例和它的闭包都不会被销毁，也是因为循环强引用：

```
paragraph = nil
```

注意， `HTMLElement` 的析构函数中的消息并没有被打印，证明了 `HTMLElement` 实例并没有被销毁。

解决闭包引起的循环强引用

在定义闭包时同时定义捕获列表作为闭包的一部分，通过这种方式可以解决闭包和类实例之间的循环强引用。捕获列表定义了闭包体内捕获一个或者多个引用类型的规则。跟解决两个类实例间的循环强引用一样，声明每个捕获的引用为弱引用或无主引用，而不是强引用。应当根据代码关系来决定使用弱引用还是无主引用。

> 注意
> Swift 有如下要求：只要在闭包内使用 `self` 的成员，就要用 `self.someProperty` 或者 `self.someMethod()` （而不只是 `someProperty` 或 `someMethod()` ）。这提醒你可能会一不小心就捕获了 `self` 。

定义捕获列表

捕获列表中的每一项都由一对元素组成，一个元素是 `weak` 或 `unowned` 关键字，另一个元素是类实例的引用（例如 `self` ）或初始化过的变量（如 `delegate = self.delegate!` ）。这些项在方括号中用逗号分开。

如果闭包有参数列表和返回类型，把捕获列表放在它们前面：

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int, stringToProcess: String) -> St
ring in
    // 这里是闭包的函数体
}
```

如果闭包没有指明参数列表或者返回类型，即它们会通过上下文推断，那么可以把捕获列表和关键字 `in` 放在闭包最开始的地方：

```
lazy var someClosure: Void -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // 这里是闭包的函数体
}
```

弱引用和无主引用

在闭包和捕获的实例总是互相引用并且总是同时销毁时，将闭包内的捕获定义为 `无主引用` 。

相反的，在被捕获的引用可能会变为 `nil` 时，将闭包内的捕获定义为 `弱引用` 。弱引用总是可选类型，并且当引用的实例被销毁后，弱引用的值会自动置为 `nil` 。这使我们可以在闭包体内检查它们是否存在。

> 注意
> 如果被捕获的引用绝对不会变为 `nil` ，应该用无主引用，而不是弱引用。

前面的 `HTMLElement` 例子中，无主引用是正确的解决循环强引用的方法。这样编写 `HTMLElement` 类来避免循环强引用：

```
class HTMLElement {
    let name: String
    let text: String?

    lazy var asHTML: Void -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

上面的 `HTMLElement` 实现和之前的实现一致，除了在 `asHTML` 闭包中多了一个捕获列表。这里，捕获列表是 `[unowned self]`，表示“将 `self` 捕获为无主引用而不是强引用”。

和之前一样，我们可以创建并打印 `HTMLElement` 实例：

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// 打印 “<p>hello, world</p>”
```

使用捕获列表后引用关系如下图所示：

这一次，闭包以无主引用的形式捕获 `self`，并不会持有 `HTMLElement` 实例的强引用。如果将 `paragraph` 赋值为 `nil`， `HTMLElement` 实例将会被销毁，并能看到它的析构函数打印出的消息：

```
paragraph = nil
// 打印 “p is being deinitialized”
```

你可以查看捕获列表 (../chapter3/04_Expressions.html)章节，获取更多关于捕获列表的信息。

上一篇：析构过程 (/project/swift/chapter2/15_Deinitialization.html)

下一篇：可选链 (/project/swift/chapter2/17_Optional_Chaining.html)

c4605

……闭包也就没法调用了，一调用程序就会报错

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
var paragraphAsHTML = paragraph!.asHTML
paragraph = nil
// paragraphAsHTML() // throw error
```

我总觉得这会让不少人被坑到

2015年12月20日 回复 顶 转发

咸鱼 (http://t.qq.com/xiaopeng50016)

翻译很好！就是看着有点晕！哈哈

(http://t.qq.com/xiaopeng50016)2015年10月28日 回复 顶(2) 转发

heronlyj (http://weibo.com/lyjonly)

终于解开了我一直以来的疑问，没有好好看书，就直接用，真实挖了好多坑啊。。。。。

(http://weibo.com/lyjonly)2015年10月22日 回复 顶 转发

土土哥 (http://weibo.com/zekunyan)

好!

(http://weibo.com/zekunyan)2015年9月26日 回复 顶 转发

lycaste (http://t.qq.com/yiyi900928)

只能说这一次翻译好多了...上一次github众筹的哪一版翻译质量不谈了....看英文版都好理解点

(http://t.qq.com/yiyi900928)

2015年8月14日 回复 顶 转发

朱利 (http://www.zhuli8.com/)

翻译的真好，辛苦了！新建的iOS开发者交流群 252839895 ，希望一起讨论学习建立一个稳定的讨论圈。

(http://www.zhuli8.com/)

2015年8月6日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

「极客学院 Wiki - wiki.jikexueyuan.com」正在使用多说 (http://duoshuo.com)

(http://

(http://